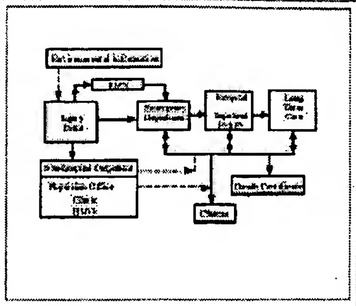
# Algorithm

From Wikipedia, the free encyclopedia

In mathematics and computer science, an **algorithm** is a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state.


Flowcharts are often used to represent algorithms.

Informally, the concept of an algorithm is often illustrated by the example of a recipe, although many algorithms are much more complex; algorithms often have steps that repeat (iterate) or require decisions (such as logic or comparison). In most higher level programs, algortihms act in complex patterns, each using smaller and smaller sub-methods which are built up to the program as a whole. In most languages are isomorphic to functions or methods.

The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers. The concept was formalized in 1936 through Alan Turing's Turing machines and Alonzo Church's lambda calculus, which in turn formed the foundation of computer science.

Most algorithms can be directly implemented by computer programs; any other algorithms can at least in theory be *simulated* by computer programs.

## Contents

## History

The word *algorithm* comes from the name of the 9th century Persian mathematician Abu Abdullah Muhammad bin Musa al-Khwarizmi. The word *algorism* originally referred only to the rules of performing arithmetic using Hindu-Arabic numerals but evolved via European Latin translation of al-Khwarizmi's name into *algorithm* by the 18th century. The word evolved to include all definite procedures for solving problems or performing tasks.

The first case of an algorithm written for a computer was Ada Byron's notes on the analytical engine written in 1842, for which she is considered by many to be the world's first programmer. However, since Charles Babbage never completed his analytical engine the algorithm was never implemented on it.

The lack of mathematical rigor in the "well-defined procedure" definition of algorithms posed some difficulties for mathematicians and logicians of the 19th and early 20th centuries. This problem was largely solved with the description of the Turing machine, an abstract model of a computer formulated by Alan Turing, and the demonstration that every method yet found for describing "well-defined procedures" advanced by other mathematicians could be emulated on a Turing machine (a statement known as the Church-Turing thesis). Nowadays, a formal criterion for an algorithm is that it is a procedure that can be implemented on a completely specified Turing machine or one of the equivalent formalisms.

## Formalization of algorithms

Algorithms are essential to the way computers process information, because a computer program is essentially an algorithm that tells the computer what specific steps to perform (in what specific order) in order to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations which can be performed by a Turing-complete system.

Typically, when an algorithm is associated with processing information, data is read from an input source or device, written to an output sink or device, and/or stored for further use. Stored data is regarded as part of the internal state of the entity performing the algorithm. The state is stored in a data structure.

For any such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will almost always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting 'from the top' and going 'down to the bottom', an idea that is described more formally by *flow of control.*

So far, this discussion of the formalization of an algorithm has assumed the premises of imperative programming. This is the most common conception, and it attempts to describe a task in discrete, 'mechanical' means. Unique to this conception of formalized algorithms is the assignment operation, setting the value of a variable. It derives from the intuition of 'memory' as a scratchpad. There is an example below of such an assignment.

See functional programming and logic programming for alternate conceptions of what constitutes an algorithm.

Some writers restrict the definition of *algorithm* to procedures that eventually finish. Others include procedures that could run forever without stopping, arguing that some entity may be required to carry out such permanent tasks. In the latter case, success can no longer be defined in terms of halting with a meaningful output. Instead, terms of success that allow for unbounded output sequences must be defined. For example, an algorithm that verifies if there are more zeros than ones in an infinite random binary sequence must run forever to be effective. If it is implemented correctly, however, the algorithm's output will be useful: for as long as it examines the sequence, the algorithm will give a positive response while the number of examined zeros outnumber the ones, and a negative response otherwise. Success for this algorithm could then be defined as eventually outputting only positive responses if there are actually more zeros than ones in the sequence, and in any other case outputting any mixture of positive and negative responses.

Summarizing the above discussion about what algorithm should consist.

- Zero or more Inputs

- One or more Outputs
- Finiteness or computability
- Definitiveness or Preciseness

### Standardised Notation

Algorithms in Computer Science suffer somewhat by the proliferation of Computer Languages each with their own syntax and basic methods. As a result, the usual standard is to write code either in C or pseudo-code. In both cases annotation usually accompanies the code in the form of an accompanying document for longer algorithms, or in the form of C style comments embedded in the code.

Some Algorithms are built for purpose and it makes sense to use a more specific language. In such cases it is usual to formalise the algorithm by defining all of it's basic functions, constants and variables before the algorithm.

### Implementation

An algorithm is a set of steps to perform a computation. Most algorithms will be implemented as computer programs. They can be expressed in any notation including English for documenting and research purposes. A more preferred way is to embody (or sometimes called *codify*) an algorithm by writing of its pseudocode. Pseudocode representation avoids ambiguities that are common in English statements. The pseudocode can also be translated into particular programming language more straightforwardly. Algorithms are implemented not only as computer programs, but often also by other means, such as in a biological neural network (for example, the human brain implementing arithmetic or an insect relocating food), in electric circuits, or in a mechanical device.

## Example

One of the simplest algorithms is to find the largest number in an (unsorted) list of numbers. The solution necessarily requires looking at every number in the list, but only once at each. From this follows a simple algorithm, which can be stated in English as

1. Let us assume the first item is largest.
2. Look at each of the remaining items in the list and make the following adjustment.
   a. If it is larger than the largest item we gathered so far, make a note of it.
3. The latest noted item is the largest in the list when the process is complete.

And here is a more formal coding of the algorithm in pseudocode:

```
Algorithm LargestNumber
  Input: A non-empty list of numbers $L$.
  Output: The largest number in the list $L$.

  largest ← $L_0$
  for each item in the list $L_{\geqslant 1}$, do
    if the item > largest, then
      largest ← the item
  return largest
```

- "←" is a loose shorthand for "changes to". For instance, with "*largest* ← the *item*", it means that the *largest* number found so far changes to this *item*.
- "**return**" terminates the algorithm and outputs the value listed behind it.

For a more complex example, see Euclid's algorithm, which is one of the oldest algorithms.

### Algorithm analysis

As it happens, most people who implement algorithms want to know how much of a particular resource (such as time or storage) is required for a given algorithm. Methods have been developed for the analysis of algorithms to obtain such quantitative answers; for example, the algorithm above has a time requirement of O(*n*), using the big O notation with *n* as the length of the list. At all times the algorithm only needs to remember two values: the largest number found so far, and its current position in the input list. Therefore it is said to have a space requirement of *O(1)*[1] (http://en.wikipedia.org/wiki/Algorithm#endnote_space). (Note that the size of the inputs is not counted as space used by the algorithm.)

Different algorithms may complete the same task with a different set of instructions in less or more time, space, or effort than others. For example, given two different recipes for making potato salad, one may have *peel the potato* before *boil the potato* while the other presents the steps in the reverse order, yet they both call for these steps to be repeated for all potatoes and end when the potato salad is ready to be eaten.

The analysis and study of algorithms is one discipline of computer science, and is often practiced abstractly (without the use of a specific programming language or other implementation). In this sense, it resembles other mathematical disciplines in that the analysis focuses on the underlying principles of the algorithm, and not on any particular implementation. The pseudocode is simplest and abstract enough for such analysis.

## Classes

There are various ways to classify algorithms, each with its own merits.

### Classification by implementation

One way to classify algorithms is by implementation means.

- **Recursion** or **iteration**: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation to other. For example, towers of hanoi is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.

- **Serial** or **parallel**: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms, which take advantage of computer architectures where several processors can work on a problem at the same time. Parallel algorithms divide the problem into more symmetrical or asymmetrical subproblems and pass them to many processors and put the results back together at one end. The resource consumption in parallel algorithms is both processor cycles on each processors and also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Recursive algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems. Those problems cannot be solved faster by employing more processors. Iterative numerical methods, such as Newton's method or the three body problem, are algorithms which are inherently serial.

- **Deterministic** or **random**: Deterministic algorithms solve the problem with exact decision at every step of the algorithm. Random algorithms as their name suggests explore the search space randomly until an acceptable solution is found. Various heuristic algorithms (see below) generally fall into the random category.

- **Exact** or **approximate**: While many algorithms reach an exact solution, approximation algorithms seek an approximation which is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

## Classification by design paradigm

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include:

- **Divide and conquer**. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively), until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in conquer phase by merging them. A simpler variant of divide and conquer is called **decrease and conquer algorithm**, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is binary search algorithm.
- **Dynamic programming**. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, we can often solve the problem quickly using *dynamic programming*, an approach that avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is, subproblems are more or less independent in divide and conquer, where as the overlap of subproblems occur in dynamic programming. The difference between the dynamic programming and straightforward recursion is in caching or memoization of recursive calls. Where subproblems are independent, there is no chance of repetition and memoization does not help, so dynamic programming is not a solution for all. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.
- **The greedy method**. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. Difference between dynamic programming and greedy method is, it extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.
- **Linear programming**. When solving a problem using linear programming, the program is put into a number of linear inequalities and then an attempt is made to maximize (or minimize) the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm. A complex variant of linear programming is called integer programming, where the solution space is restricted to all integers.
- **Reduction**: It is another powerful technique in solving many problems by transforming one problem into another problem. For example, one selection algorithm for finding the median in an unsorted list is first translating this problem into sorting problem and finding the middle element in sorted list. The goal of reduction algorithms is finding the simplest transformation such that complexity of reduction algorithm does not dominate the complexity of reduced algorithm. This technique is also called *transform and conquer*.
- **Search and enumeration**. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes the search algorithms and backtracking.
- **The probabilistic and heuristic paradigm**. Algorithms belonging to this class fit the definition of an algorithm

more loosely.

1. Probabilistic algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness.
2. Genetic algorithms attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and "survival of the fittest". In genetic programming, this approach is extended to algorithms, by regarding the algorithm itself as a "solution" to a problem. Also there are
3. Heuristic algorithms, whose general purpose is not to find an optimal solution, but an approximate solution where the time or resources to find a perfect solution are not practical. An example of this would be local search, taboo search, or simulated annealing algorithms, a class of heuristic probabilistic algorithms that vary the solution of a problem by a random amount. The name "simulated annealing" alludes to the metallurgic term meaning the heating and cooling of metal to achieve freedom from defects. The purpose of the random variance is to find close to globally optimal solutions rather than simply locally optimal ones, the idea being that the random element will be decreased as the algorithm settles down to a solution.

### Classification by field of study

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are search algorithms, sort algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, computational geometric algorithms, combinatorial algorithms, machine learning, cryptography, data compression algorithms and parsing techniques.

*See also:* **List of algorithms** for more details.

Some of these fields overlap with each other and advancing in algorithms for one field causes advancement in many fields and sometimes completely unrelated fields. For example, dynamic programming is originally invented for optimisation in resource consumption in industries, but it is used in solving broad range of problems in many fields.

### Classification by complexity

Some algorithms complete in linear time, and some complete in exponential amount of time, and some never complete. One problem may have multiple algorithms, and some problems may have no algorithms. Some problems have no known efficient algorithms. There are also mappings from some problems to other problems. So computer scientists found it is suitable to classify the problems rather than algorithms into equivalence classes based on the complexity.

*See also:* **Complexity classes** for more details.

## Legal issues

Some countries allow algorithms to be patented when embodied in software or in hardware. Patents have long been a controversial issue (see, for example, the software patent debate).

Some countries do not allow certain algorithms, such as cryptographic algorithms, to be exported from that country.

## Examples of useful simple algorithms

## See also

- Algorism

- Approximation algorithms
- Data structure
- Randomized algorithm
- Timeline of algorithms
- Wikibooks:Algorithms

## Notes

^ Although in this example the size of the numbers itself is unbounded, one could therefore argue that the space requirement is O(log $n$), in practice, however, the space taken up by a number is fixed.

## References

- Important algorithm-related publications

## External links

- Paul E. Black, algorithm (http://www.nist.gov/dads/HTML/algorithm.html) at the NIST Dictionary of Algorithms and Data Structures.
- Gaston H. Gonnet and Ricardo Baeza-Yates: Example programs from *Handbook of Algorithms and Data Structures.* (http://www.dcc.uchile.cl/~rbaeza/handbook/) Free source code for many important algorithms.
- Dictionary of Algorithms and Data Structures (http://www.nist.gov/dads/). "This is a dictionary of algorithms, algorithmic techniques, data structures, archetypical problems, and related definitions."
- Numerical Recipes (http://www.nr.com/)
- Computers/Algorithms @ dmoz.org (http://dmoz.org/Computers/Algorithms/)
- *Musicalgorithms* (http://musicalgorithms.ewu.edu/) An interesting way of using algorithms to make music.
- The Algorithmist (http://www.algorithmist.com/) is a web site dedicated to algorithms.

Retrieved from "http://en.wikipedia.org/wiki/Algorithm"

Categories: Articles with sections needing expansion | Algorithms | Arabic words | Discrete mathematics | Mathematical logic